Timeline: An Operating System Abstraction for Time-Aware Applications

Fatima M. Anwar *‡, Sandeep D'souza †‡, Andrew Symington *‡, Adwait Dongare †‡,
Ragunathan (Raj) Rajkumar †, Anthony Rowe †, Mani B. Srivastava *
*Department of Electrical Engineering, University of California Los Angeles, Los Angeles, CA
†Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA

***Abstract*—Having a shared and accurate sense of time is critical to distributed Cyber-Physical Systems (CPS) and the Internet of Things (IoT). Thanks to decades of research in clock technologies and synchronization protocols, it is now possible to measure and synchronize time across distributed systems with unprecedented accuracy. However, applications have not benefited to the same extent due to limitations of the system services that help manage time, and hardware-OS and OS-application interfaces through which timing information flows to the application. Due to the importance of time awareness in a broad range of emerging applications, running on commodity platforms and operating systems, it is imperative to rethink how time is handled across the system stack. We advocate the adoption of a holistic notion of Quality of Time (QoT) that captures metrics such as resolution, accuracy, and stability. Building on this notion we propose an architecture in which the local perception of time is a controllable operating system primitive with observable uncertainty, and where time synchronization balances applications' timing demands with system resources such as energy and bandwidth. Our architecture features an expressive application programming interface that is centered around the abstraction of a *timeline* – a virtual temporal coordinate frame that is defined by an application to provide its components with a shared sense of time, with a desired *accuracy* and *resolution*. The timeline abstraction enables developers to easily write applications whose activities are choreographed across time and space. Leveraging open source hardware and software components, we have implemented an initial Linux realization of the proposed *timeline*-driven QoT stack on a standard embedded computing platform. Results from its evaluation are also presented.**

I. Introduction

Maintaining a shared notion of time is critical to the performance of many distributed and cyber-physical systems such as sensor networks, swarm robotics [1], high frequency trading [2], tele-surgery [3], Big Science [4] and global-scale databases [5]. Technologies such as GPS, Precision Time Protocol [6] and chip-scale atomic clocks have made it possible to provide systems with accurate, stable, and a common notion of time across a network. However, other technology trends have made it harder for applications to benefit from these advances in timing technologies. For example, asymmetric medium delays degrade time transfer [7], imperfect oscillators cause timing jitter [8], multi-core systems have timing inconsistencies [9], and abstractions like virtual machines introduce greater timing uncertainty [10]. In multiple domains ranging from database consistency [5] to interactive cloud gaming [11], the knowledge of timing uncertainty has proven to be useful. However, application-level visibility into timing uncertainty remains largely unexplored in current systems.

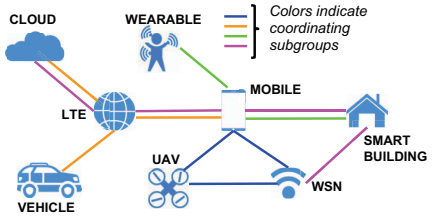


Fig. 1: Coordinating subgroups in the Internet of Things (IoT) require access to a shared sense of time

We introduce the notion of *Quality of Time (QoT)* as the end-to-end uncertainty in time delivered to a user by the system. Our goal is to create a QoT architecture which reacts to application timing demands, and exposes timing uncertainty to applications in an easy-to-use, secure and scalable way. Our architecture treats time as a controllable, verifiable and observable primitive thus delivering the requested QoT to applications.

Technologies such as the Internet of Things drive us towards a world where a large number of end-point and intermediary devices share, store and manipulate real-time data. This has led to the emergence of time-aware applications with extremely diverse timing requirements. The timing precision and accuracy requirements of applications in one domain of connected systems may be substantially different for others, and may also change over time. Figure 1 illustrates this diversity by showing a distributed system with many different device types and communication channels forming timing subgroups (shown in different colors). These devices are often heterogeneous and have different resource constraints. Traditionally, time synchronization has always been best-effort, agnostic to application requirements and wasteful of system resources.

We advocate for *factored coordination* where subsets of coordinating nodes synchronize their clocks to only the desired accuracy. To achieve this objective, we propose the *timeline* abstraction for managing and synchronizing time in computing systems. Time-aware applications express their willingness to share common time by binding to a virtual timeline, rather than to a specific device or time coordinate system. Each

‡Equally contributing authors

application specifies an accuracy (the deviation an application is willing to tolerate from the timeline) and a resolution (an upper bound on the minimum discrete time tick) with which it binds to the timeline). The key advantage of this approach is that a device can support multiple timelines, with different and time-varying accuracy requirements. Thus enabling the paradigm of factored coordination. Furthermore, timelines provide scalability, robustness to node failures as well as dynamic binding and unbinding.

The overview of a timeline-driven QoT architecture is shown in Figure 2. The architecture closes the loop between the timing requirements of applications, and how well the system is able to meet their needs by propagating timing uncertainty back to the applications. It characterizes the timekeeping hardware capabilities e.g. oscillators and timestamping mechanisms and exposes them as controllable and disciplinable clocks. The system can adjust these clocks and/or switch between them to balance application needs with the system resources. For example, a system can switch from a high energy rubidium oscillator to an unstable low energy quartz crystal oscillator, from hardware timestamping to software timestamping, from a high synchronization rate to a low rate to balance system resources with application QoT requirements.

The primary contributions of our work are as follows:

- We propose the notion of Quality of Time based on timing uncertainty.
- We investigate how time-aware applications and the OS should exchange information about time and develop a model that describes how (i) applications interact with a shared notion of time, (ii) applications register their timing requirements with the OS and (iii) how timing uncertainty is conveyed from the OS to applications.
- We propose a platform-independent OS abstraction called a *timeline*, and present an application programming interface (API) that greatly simplifies the development of QoT-aware choreographed applications.
- We provide an end-to-end timeline-driven QoT architecture and its corresponding implementation for Linux.
- We conduct a series of micro-benchmarks to verify the performance of our QoT architecture on a Linux-based embedded platform, the Beaglebone Black [12].

II. Related Work

The notion of time uncertainty is not new to the field of time synchronization. NTP [13] computes a bound on time for every timestamp sample and applies clock filtering algorithms to filter out the false samples. However, this bound is never exposed to applications and hence becomes invalid when a clock adjustment is made. Google Spanner [5] utilizes the True Time API to show how the knowledge of uncertainty in time can be used to achieve the external consistency of transactions in a global database. However, Spanner is a closed system and the TrueTime API are tailored only to database transactions. Additionally, both the TrueTime API and POSIX API do not treat the notion of time as an application-specified requirement. In contrast, our work seeks to provide a universal framework

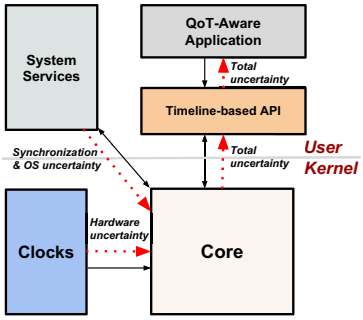


Fig. 2: Timeline-driven Quality-of-Time Architecture

with associated timeline-based API that lets applications specify their QoT requirements, and also exposes the achieved QoT to applications for varied uses including coordination and adaptation.

At the programming level, time-triggered and event-triggered computation models provide timing determinism. The time-triggered architecture (TTA) [14] addresses issues in real-time programming by establishing a global time-base to specify interaction among nodes, whereas event-triggered architectures like Ptides [15] map model time to real time, only when systems interact with the physical world, e.g. using sensors and actuators. PtidyOS [16] is a microkernel that generates target specific code for the Ptides model. Our timeline-driven QoT architecture utilizes a different paradigm. It focuses on assisting coordinated and distributed time-aware applications on local as well as wide area networks, through a hardware-independent system library as well as an entire framework built using commodity hardware and software.

Also relevant and complementary to our work is research in time synchronization on the analytical modeling of clock uncertainties [17], and methods to compensate for them via approaches such as Kalman filtering [18].

III. Timelines

Modern operating systems generally derive their notion of time from the highest-quality timer available on a system. For example, in Linux, multiple virtual clocks are derived from such a single timer. These virtual clocks all share the same accuracy and resolution, and expose themselves to applications in userspace via the standardized POSIX clock [19] interface. The Linux kernel allows users to precisely schedule events on any of these clocks notion of time by using the underlying High Resolution Timer subsystem. However, Linux does not expose the uncertainty in its knowledge of time to applications, nor does it allow applications to specify or change their QoT requirements.

Linux and other OSs allow clocks to be disciplined using synchronization techniques such as NTP [13] and PTP [6]. These and other synchronization protocols are based on the traditional "trickle-down time" approach (as shown in Figure 3a), where a master (statically defined or dynamically chosen) sits on top of a timing hierarchy, restricting other nodes to be slaves. The slave nodes measure time with respect to the master's time axis. Furthermore, messages are exchanged at

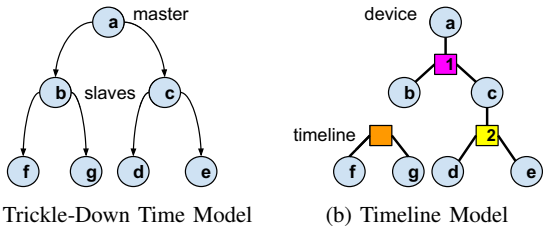
a pre-calibrated rate corresponding to the tightest accuracy requirement in the network. Such approaches are inherently centralized, not adaptive, and are wasteful of resources.

We introduce an alternative paradigm based on *factored coordination* that abstracts away from "trickle-down time", and only coordinating nodes synchronize their time. To enable adaptive time management and factored coordination, we propose a new timing abstraction called a *timeline*. A timeline is a virtual reference time base that is completely decoupled from any specific reference device or time system. It enables developers to easily implement coordinated applications. Applications that need to coordinate their tasks, bind to a common timeline and synchronize their time as shown in Figure 3b. For example, node f and g bind to Timeline 3 and synchronize their time with each other rather than the entire network. This enables us to escape the limitations of the traditional master-slave synchronization paradigm and support applications that dynamically bind and unbind from timelines as needed. As a result, our system is distributed, enables adaptive groupings and balances timing requirements with system resources.

A timeline's instantaneous reference could be any node in the network, and can be passed adaptively between nodes, based on available resources or network conditions. For example, multiple players in the same locality playing a virtual reality game need not synchronize to some external server, instead they can join a timeline and synchronize with one another for their timing needs.

Applications define their timing requirements with respect to a timeline, and not any specific master node. When an application binds to a timeline, it specifies its timing requirements in terms of an $accuracy$ and a $resolution$, collectively referred to as a $binding$. This is shown in Figure 4. The accuracy of a timeline is an asymmetric interval around the reference time that determines the maximum error an application can tolerate in its time estimate. The resolution of a timeline is the minimum tick requirement of an application. Both accuracy and resolution are represented by a $\{second,\ attosecond\}$ tuple with each attribute being a 64-bit unsigned integer. For example, if some applications want to coordinate their sensing tasks in the order of a millisecond, and they require nanosecond granularity in their time ticks, they will all bind to the same timeline with an accuracy of 1 millisecond, $\{0,\ 1e15\}$, and a resolution of 1 nanosecond $\{0,\ 1e9\}$.

The accuracy and resolution of a timeline affect the system's choice of switching between different clock sources or oscillators with varying frequencies and stability. High accuracy



(a) Trickle-Down Time Model (b) Timeline Model

Fig. 3: Traditional v/s Timeline based synchronization models

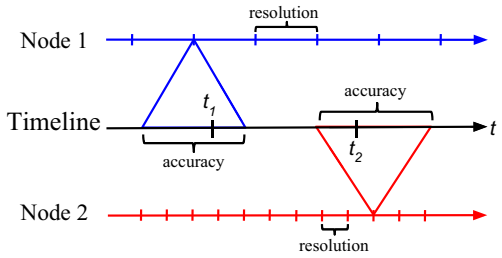


Fig. 4: Node 1 and 2 bind to a timeline with a desired binding accuracy, and binding resolution. Note that accuracy is an asymmetric interval around true time.

necessitates the use of a stable clock, whereas high resolution requires a high-frequency clock. Clocks with greater stability and higher resolution generally consume more energy. This abstraction provides an opportunity to balance application needs (accuracy and resolution) with resources. Motivated by the timeline abstraction, we next describe our QoT architecture and its accompanying system stack.

IV. QoT Architecture

The QoT architecture uses the timeline abstraction to make uncertainty in time observable and controllable across a broad range of QoT-Aware applications. The QoT architecture is comprised of three distinct components: (i) *Clocks*, (ii) *System Services*, and (iii) *QoT Core*. In this section, we describe the individual components of the QoT architecture (as shown in Figure 2), and show how they interact with each other.

A. Clocks

The QoT architecture exposes timekeeping hardware as Clocks, which play a major role in delivering knowledge of time with associated QoT to the applications. Based on the functionality provided, we categorize them into two types:

Core Clocks drive all the functionality of the stack. All timelines derive their reference time as a projection from a core clock. For a clock to qualify as a core clock, it must provide (i) the ability to read a *strictly-monotonic* counter, which cannot be altered by any system process, (ii) the ability to schedule events along a timeline reference, and (iii) provide the hardware resolution and uncertainty associated with reading the clock. Optionally, a core clock may also expose interfaces to timestamp and generate external events.

Network Interface Clocks (NICs) assist in disciplining the local time to some global reference time. Only those network interfaces which have the ability to accurately timestamp network packet transmission and reception, at the physical layer, are exposed as NICs. This enables precise calculation of the offset between two clocks, and the propagation delay associated with a medium. A NIC is similar to a core clock in providing the ability to read time, and optionally provide I/O functionality for precisely timestamping an event, or generating a very deterministic pulse in the future. A NIC, however, differs from a core clock in that: (i) it is not necessarily monotonic, (ii) it may be disciplinable and (iii) it does not

provide the ability to generate interrupts. Hence, it cannot be used to accurately schedule user-level application threads.

Our architecture supports these two clock categories, and provides mechanisms to synchronize them with each other. While every node must contain a core clock, it is not necessary for a node to contain a NIC.

B. System Services

System services are user-space processes responsible for distributing timeline metadata, quantifying timing uncertainties, and synchronizing time within and across nodes.

1) Data Distribution Service (DDS): DDS [20] is a networking middleware service which simplifies networking programming for our architecture. It provides a publish-subscribe framework, which collects all the timeline requirements and gives participating nodes the ability to decide the reference time in a decentralized fashion.

2) Synchronization Service: Modern OSs expose only a single clock and synchronize it on a best-effort basis by being oblivious to the application requirements. In contrast, our timeline-driven architecture supports multiple timelines on a single node, each having its own notion of time. In Figure 3b, node c is part of both Timelines 1 and 2. Hence, it runs two parallel instances of the synchronization service for two different timelines, each achieving only the desired accuracy.

A timeline represents a mapping from a *local* core time to a *global* reference time. To generate this mapping we require a two-step synchronization procedure as shown in Figure 5. In the first step, we synchronize the NIC to the core clock on a single node. Once the NIC is aligned to core time all timestamps provided by a NIC can be mapped to core time.

In the second step, we perform *inter-node* synchronization. NICs exchange synchronization packets across nodes and timestamp them in core time, thus generating a timeline mapping using a first-order linear model, $t_M = t_M^{'} + (1 + ppb/10^9) * (t_c - t_c^{'})$, where t_M is the current timeline mapping that is derived from previous mapping $t_M^{'}$, frequency bias ppb (parts per billion) and current and previous core time, t_c and $t_c^{'}$ respectively. In Figure 5, multiple timelines are maintained as logical mappings and they provide the ability to synchronize with multiple nodes with totally different accuracy requirements, thus enabling the factored coordination paradigm. The Linux PTP Project [21] also runs two-step synchronization. It first aligns NICs across different nodes, and then it synchronizes the system clock to the NIC. However, this approach does not scale to multiple timelines. Hence, we keep the core clock strictly monotonic and maintain multiple timelines as logical mappings from the core clock.

Synchronization Uncertainty: Time synchronization performance is limited by various stochastic delays in the system: *Propagation delay*, *Transmit delay*, *Receive delay* and *Residency delay*. These delays introduce uncertainty in our time measurements. Using a statistical approach we can calculate the upper bound on synchronization uncertainty as, $E^U = \{(ppb_m - ppb + \Delta ppb)/10^9\} * (t_c - t_c^{'}) + (e_m + \Delta e)$, and the lower bound on uncertainty, $E^L = \{(ppb - ppb_m + \Delta ppb)/10^9\} * (t_c - t_c^{'}) + (e_m - \Delta e)$, where ppb_m is the mean bias, Δppb is the standard deviation of bias, e_m is the mean offset and Δe is the standard deviation of offset. Instead of using only local statistical information to calculate the bounds, network wide information using a Kalman filter can also be applied to calculate tighter uncertainty bounds. Whichever model is used for the uncertainty bound calculation, a global time estimate at any point in time should be $t_M - E^L < t_M < t_M + E^U$.

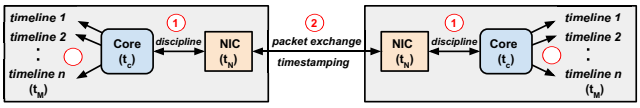


Fig. 5: End-to-end time synchronization using timelines. (1) NIC is disciplined by the core clock on a single node ($t_N = t_C$). (2) NICs on two nodes exchange packets and timestamp them in core time. (3) The timestamps are used to work out the Core – Core mapping (t_M) and are stored in the form of a logical timeline mapping.

3) System Uncertainty Estimation Service: Every timestamp read by a user application contains an uncertainty value introduced by the OS, which is a function of factors like the system load and CPU operating frequency. This service continuously updates these uncertainty statistics and passes it to the the stack. These uncertainty values are appended to every timestamp as an uncertainty bound.

C. QoT Core

The QoT Core (also referred to as the *core*) acts as a bridge between all the stack components, and the host OS. The core performs a range of functionality:

Timeline Management: To satisfy different QoT requirements, the core keeps track of different timelines and their associated bindings, and handles their creation and destruction. It also provides an interface for applications to bind to a timeline and specify their QoT requirements.

Clock Management: The core provides an interface for different hardware clocks to register with it, and exposes an interface for a *privileged* user to choose and switch between these different clocks. The core utilizes this chosen clock to maintain a monotonic sense of time, referred to as *core time*. The key idea is that a privileged daemon should be able to automatically select the core clock in a manner that balances clock stability/resolution with energy consumption. The core also maintains the projection parameters from the core clock to each timeline reference, and provides an interface for the synchronization service to manipulate them.

Event Scheduling: Scheduling an application on a global notion of time is important to execute distributed tasks synchronously. Hence, the core provides applications the ability to *synchronously* schedule events based on a timeline reference. The core provides this functionality in the form of *timed waits* by interfacing with the OS scheduler. Timed waits provide threads the ability to sleep for a relative duration or until an absolute time. The scheduling subsystem is also designed to

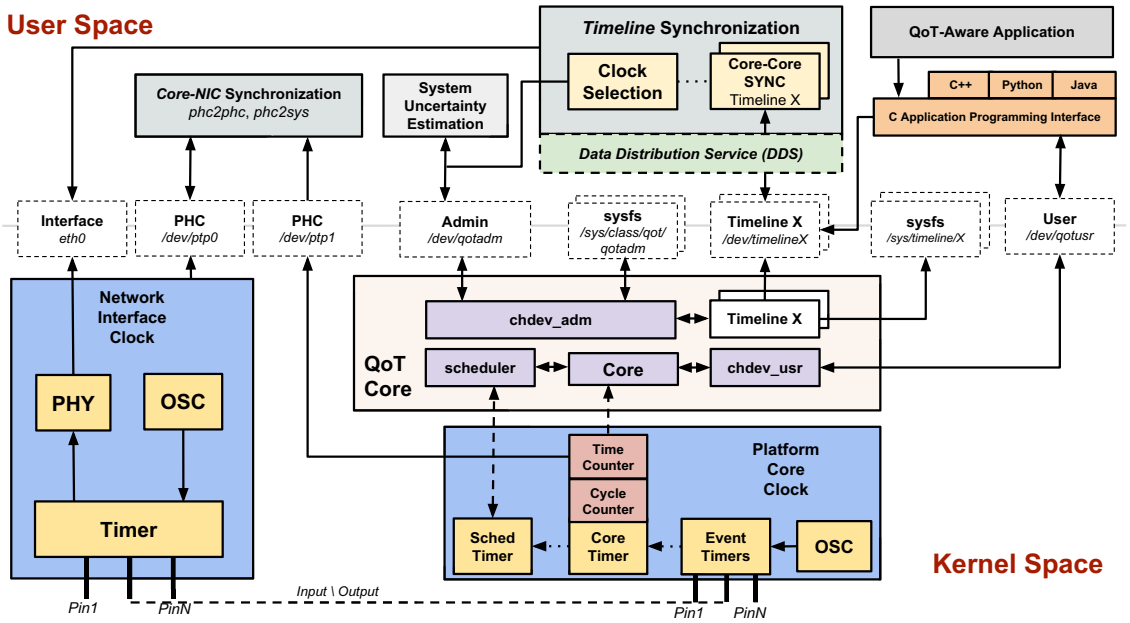


Fig. 6: The QoT Stack for Linux

dynamically compensate for any synchronization changes to a timeline reference.

QoT Propagation: One of the goals of the QoT stack is to expose the timing uncertainty to applications, so that the framework gives the current estimate of time, along with the uncertainty associated with it. As shown in Figure 2, the core propagates the uncertainties from different stack components, appending uncertainty to every time estimate. It also provides interfaces for the system services and the hardware clock to expose/update these uncertainty values.

V. QoT Stack for Linux

To demonstrate the possibility of our QoT architecture, we have developed a timeline-driven QoT Stack for Linux. Given the variety of supported software and hardware, Linux is an ideal candidate to prototype a cross-platform timeline abstraction. Since, every system has its unique timing limitations, we attempt to quantify and work with them, instead of forcing the use of a particular platform.A complete architectural diagram of our QoT Stack for Linux appears in Figure 6. We adopt a modular design to avoid requiring changes to the Linux kernel, instead relying on loadable kernel modules and userspace daemons to make Linux QoT-aware.

A. Timelines in Linux

Our Linux-based prototype implements a timeline as a `/dev/timelineX` character device, where X corresponds to a unique identifier. The character device exposes the *timeline reference* as a POSIX clock [19] to userspace, which is disciplinable by a synchronization service. The timeline character device also exposes an Input-Output Control (`ioctl`) interface, for applications to bind/unbind to a timeline, specify/update their QoT requirements, and read the timeline's reference time with an uncertainty estimate. In the Linux kernel, timelines are stored and ordered on a red-black tree which provides an $O(log(N))$ look-up time with a string identifier. For details on timelines and bindings data structures, please refer to Appendix A.

B. Clocks

Clocks (shown as *Network Interface Clock* and *Platform Core Clock* in Figure 6) are managed via drivers and use the Linux **`ptp_clock`** libraries to abstract away from architecture-specific sources. This abstraction provides the ability to enable or disable the clock source, configure timer pins (for timestamping inputs or pulse-width modulated outputs) and discipline the external clock (either in hardware or software). Pins are configured through the hardware timer subsystem using **`.enable`** and **`.verify`** function callbacks. The time can be observed or set through **`.gettime64`** and **`.settime64`** function callbacks. The kernel drivers implement the correct function callbacks, and register the existence of the precise clock through **`ptp_clock_register`**, with the kernel's PTP subsystem. In PTP terminology, these clocks are referred to as *Precise Hardware Clocks* (PHC), which is any clock or network interface that supports hardware timestamping and GPIO capabilities. These clocks are exposed to userspace as /dev/ptpX character devices and they register their capabilities, uncertainties, resolution and function hooks with the core module.

C. System Services

Data Distribution Service: OpenSplice [20] is used as the data distribution service in our stack. It disseminates timeline metadata across the entire network. Once every node has a complete picture of *timelines* on all nodes, they compete for providing the reference time and in our baseline implementation, the node with the highest accuracy requirement is chosen

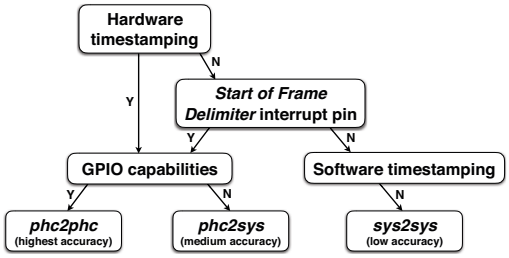


Fig. 7: Decision tree for choosing a time synchronization service based on hardware capabilities

to provide the reference time to the timeline's subgroup. The synchronization rate is determined by the highest accuracy requirement in the network. Hence, the node which has the highest requirement in its timing subgroup can become a master and push packets with a rate corresponding to its accuracy requirement.

Synchronization Service: The synchronization service operates in userspace and comprises of *Core-NIC Synchronization* and *Timeline Synchronization* daemons, as shown in Figure 6. The Timeline Synchronization daemon is implemented by patching the Linux PTP Project [21]. It calculates clock discipline parameters, and disciplines the `/dev/timelineX` character devices through the **`.settime`** and **`.adjtime`** POSIX clock APIs. The mappings are stored in the kernel so that the timeline reference can be easily returned using the **`.gettime`** POSIX clock API. A detailed description of PTP can be found in Appendix C.

We also create a synchronization service *phc2phc* that aligns two *Precise Hardware Clocks* (PHC): clocks which support hardware timestamping and GPIO with external hardware timestamping, and deterministic hardware interrupt capabilities. Our implementation performs *Core-NIC* synchronization using *phc2phc*. If one of the clock is not a PHC, we use the *phc2sys* [21] service to synchronize clocks. The decision tree in Figure 7 shows how timestamping and GPIO capabilities of a clock influence our choice of synchronization service. Certain network interfaces do not support hardware timestamping, but provide a hardware interrupt upon the Start of Frame Delimiter (SFD) of a synchronization packet. In this case, if the core clock is a PHC, it can timestamp the SFD interrupt in hardware and run *phc2phc* across multiple nodes for high accuracy. However, certain network interfaces neither expose a PHC, nor support SFD. In this case, the core clocks resort to software time stamping and perform *sys2sys*. Table I lists some example network interfaces with different hardware capabilities and the corresponding synchronization service.

TABLE I: Network Interface Capabilities

NIC	Capabilities	Service
TI CPSW	PHC, GPIO interrupt	*phc2phc*
AT86RF233	PHC, SFD interrupt	*phc2phc*
DW1000	PHC, SFD interrupt	*phc2phc*
IEEE 802.11	None	*sys2sys*

System Uncertainty Estimation Service: This service tries to get a probabilistic estimate of the OS clock read uncertainty by reading the core clock in a tight loop from userspace, via a privileged interface (`/dev/qotadm`). By taking the difference of consecutive timestamps, the service calculates the uncertainty distribution.

D. Linux QoT Core Kernel Module

The Linux QoT Core, shown as the central component in Figure 6, is implemented as a loadable kernel module. It consists of the following sub-modules.

Scheduler Interface: Each active timeline maintains a red-black tree of waiting threads, ordered by their wake-up times in the timeline reference. When an application thread issues a timed wait request, the thread is suspended and en-queued on a red-black tree corresponding to the timeline to which it is bound. Waking up applications from their suspended state relies on the interrupt functionality of the core clock. When the callback triggers, the interrupt handler checks each active timeline for tasks that need to be woken up, and moves such tasks from the wait queue to the ready queue. Subsequently, the task is scheduled as per its priority, and the policy being used by the scheduler. This introduces scheduling uncertainty, as other threads may also be present on the ready queue. Before the task is actually scheduled, the core returns a timestamp of the scheduling instant along with an uncertainty estimate. This enables an application to take a decision, based on the received QoT. The scheduling policy *agnostic* design, enables the stack to be portable to a range of different Linux kernels, and prevents it from being tied down to a specific kernel version. It also gives the opportunity for OS developers to use scheduling policies best suited for the target platform. Future implementations of the stack will include techniques to probabilistically compensate for the scheduling uncertainty.

Decisions on waking up a task, or programming the next interrupt callback, rely on the projections between core time and the timeline references. The scheduling interface compensates for any synchronization changes to these projections. When a synchronization event occurs, the interface checks the head of the timeline queue, to decide whether the change in the projection, necessitates a task to be scheduled earlier than previously estimated.

User Interface: The core exposes a set of thread-safe `ioctl` interfaces in the form of a character device, `/dev/qotusr`, to userspace. It gives user applications the ability to create/destroy a timeline, read timestamps with uncertainty estimates, as well as issue timed waits on a timeline reference. The user interface also provides applications the ability to access the external timestamping and event triggering functionality of the core clock (if supported by hardware).

Admin Interface: This is a special character device `/dev/qotadm`, which enables a privileged daemon to control specific parameters of the QoT stack. It provides an `ioctl` interface, which allows a privileged user to get information on clocks, switch between different core clocks, as well as get/set the OS uncertainty associated with reading timestamps.

Sysfs Introspection: The core provides a `sysfs` interface for a user to view and change the state of the system using

file operations. It can be used to develop complex visualization tools or to integrate with existing monitoring systems.

On the Beaglebone Black platform, the memory footprint of the QoT Stack for Linux is 4.071 MB. The current implementation re-implements a number of existing components for easier debugging, leading to a large code size. Future implementations will focus on optimization.

VI. Application Programming Interface

We provide an API that allows programmers to simplify the development of distributed QoT-Aware applications. The key API calls can be found in Table II in the Appendix, and are categorized by their functionality: (i) *Timeline Association* APIs allow applications to bind/unbind to a specific timeline, and specify/update their QoT requirements. (ii) *Time Management* APIs allow applications to read the timeline notion of time with the uncertainty estimate. (iii) *Event Scheduling* APIs allow applications to schedule events using timed waits on the timeline reference, along with returning the uncertainty as when the event was actually scheduled. Additionally, the APIs also provide the ability to trigger events at a deterministic point in the future, as well as accurately timestamp external events, contingent on hardware support from the core clock.

Listing 1: QoT-Aware TDMA Application

```
name = "tdma-timeline"; /* Timeline UUID */
/* Timeline accuracy equivalent to TDMA guard band */
timeinterval_t accuracy = {
  .below = TL_FROM_nSEC(0),
  .above = TL_FROM_nSEC(TDMA_GUARD_BAND),
};
/* Timeline resolution equivalent to TDMA period */
timelength_t res = TL_FROM_nSEC(TDMA_PERIOD);
timelength_t period = TL_FROM_nSEC(TDMA_PERIOD);
timepoint_t start_offset
  = TP_FROM_nSEC(get_my_slot()*TDMA_SLOT_LENGTH);
/* Bind to a timeline with requested UUID */
timeline_t timeline = timeline_bind(name, accuracy, res);
/* Set period and start offset */
timeline_setschedparams(timeline, period, start_offset);
/* Periodic TDMA Transmission */
while(tdma_running) {
  /* Sleep until start of next transmit slot */
  status = timeline_waituntil_nextperiod(timeline);
  if(status == QOT_OK) {
    /* Transmit if uncertainity within bound */
    transmit_packet(message);
  } else {
    hold_off();
  }
}
timeline_unbind(timeline); /* Unbind from a timeline */
```

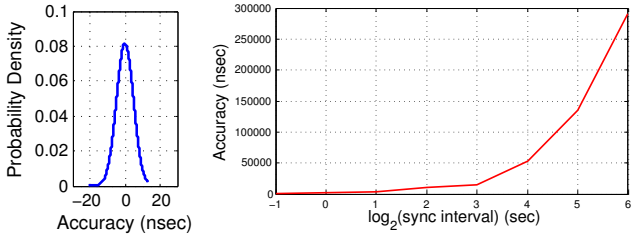
Using these APIs, we present a sample code snippet for a *Time Division Multiple Access (TDMA)* application in Listing 1. In this implementation, multiple nodes need to be allocated transmit slots, such that no packet collisions occur. It is essential that all nodes have a shared notion of time, along with the uncertainty associated with it. The application compensates for synchronization errors using guard bands. If timing uncertainty increases beyond these guard bands (e.g. if synchronization is lost), then packets will collide. Notifying the application about the returned QoT gives it the ability to *adapt*.

Such an application would start by creating a binding to a timeline, with desired accuracy and resolution using **`timeline_bind`**. The detail of data types for timeline and it's binding requirement – accuracy and resolution – can be found in Appendix B. Given that transmitting in a TDMA slot is inherently periodic, the application can set its period and start offset using **`timeline_setschedparams`**. Subsequently, the application executes a loop, where it calls **`timeline_waituntil_nextperiod`**, which wakes up the task every period, using the programmed parameters. This call returns a status whether the returned QoT is within the accuracy requested by the application. The application can make use of this information to take a decision on transmitting a packet. Finally, before the application terminates, it unbinds from the timeline using **`timeline_unbind`**. We also contrast our QoT-aware TDMA application with one written using the Linux POSIX API. The Linux API based TDMA application does not have a notion of QoT and cannot provide end-to-end estimates of timing uncertainty. The corresponding code snippet along with a detailed comparison can be found in Listing 5 in Appendix D.

VII. Experimental Evaluation

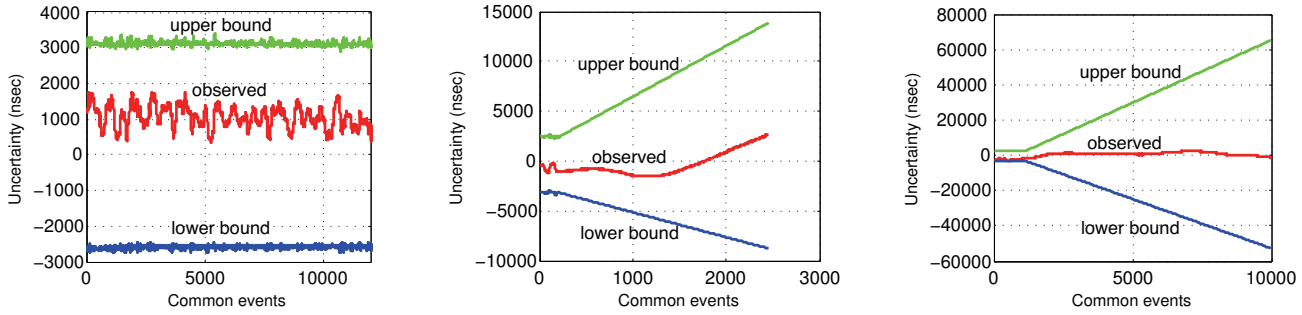
Our prototype stack provides hardware support for the popular Beaglebone Black (BBB) embedded Linux platform [12]. We implement drivers to support the Texas Instruments (TI) AM335x ARM Cortex-A8 System-on-Chip (SoC) found on the BBB. The SoC supports the IEEE 1588 standard [6] (Precision Time Protocol) over Ethernet, and has the ability to timestamp network packets at the physical layer. The drivers serve as a reference implementation, and provide core concepts which can be ported to a variety of platforms. Corresponding to the two types of clocks that we defined in Section IV-A, we have implemented platform-specific drivers whose details can be found in Appendix E.

Our testbed comprises multiple BBB nodes, with the Linux 4.1.12-rt kernel, connected via an IEEE 1588-compliant switch [22], running the *synchronization service* which is a patched version of the Linux PTP Project [21]. Instead of disciplining the ethernet controller's NIC on the node (`/dev/ptp0`), our service supports simultaneous synchronization of multiple timelines, and disciplines the `/dev/timelineX` character devices. Using this testbed, we now present multiple micro-benchmarks which demonstrate the ability of our stack to perform synchronization, expose uncertainty and perform *choreographed* scheduling.



(a) Core-NIC (b) Accuracy v/s Synchronization Interval

Fig. 8: (a) Core-NIC synchronization accuracy (b) Illustrating the adjustable synchronization parameter



(a) Synchronization on for 1 hour (b) Synchronization on, then off for 5 mins (c) Synchronization on, then off for 1 hour

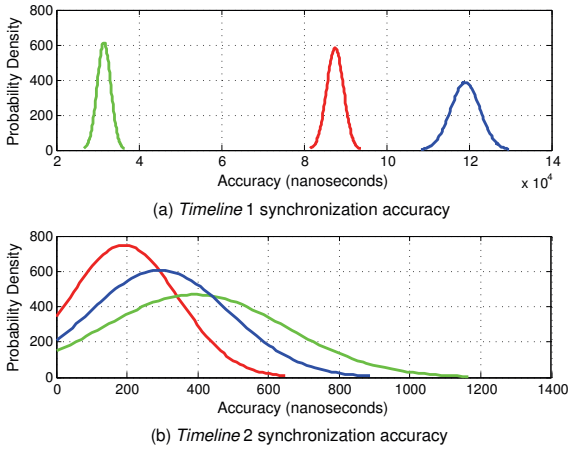
Fig. 9: Upper and lower bounds around the observed uncertainty with and without synchronization. Note the change in y-axis scale, increasing from (a) to (c)

A. *Synchronization Uncertainty*

The prerequisite for end-to-end synchronization – mapping *local* core time to a *global* timeline reference – is to first synchronize the on-board NIC with the local core clock. We use a programmable hardware timer on the BBB AM335x to trigger very deterministic and periodic outputs on a pin in core time, which is then timestamped by the NIC. The difference between the core and NIC timestamps is used to work out the clock disciplining parameters. We plot the distribution of this difference in Figure 8a, which indicates the Core-NIC synchronization accuracy, which is in the order of nanoseconds. A similar approach can work on other hardware platforms as well.

The ability to control the synchronization accuracy, is a key goal of the QoT stack. We use the transmission rate of synchronization packets as a control knob to adjust the accuracy, and the resulting plot is shown in Figure 8b. Note that increasing the synchronization packet transmission rate reduces synchronization error and increases timing accuracy. This proves the existence of such adjustable parameters, which can be exposed to the userspace services so that they can control the system performance and meet the QoT requirements.

Now that we have synchronized the NIC to the core clock and established the relationship between synchronization rate and accuracy, we use a topology similar to the one in Figure 3b for end-to-end synchronization. There are two timing subgroups: nodes *a*, *b* and *c* bound to Timeline 1 with an accuracy of 100 μsec; and nodes *c*, *d* and *e* bound to Timeline 2 with an accuracy of 1 μsec. The system sets a synchronization rate of 0.05 Hz for Timeline 1, and 2 Hz for Timeline 2 according to their accuracy requirements. We conducted experiments on this topology to demonstrate that the QoT stack runs multiple and parallel synchronization sessions on a single node, which disciplines multiple timelines simultaneously. The results are shown in Figure 10, where node *c* maintains two timelines with very different accuracy requirements of 100 μsec and 1 μsec, with respect to Timelines 1 and 2 respectively. This validates our initial claim that the timeline-driven architecture not only supports multiple virtual time references on a single node, but also synchronizes only to the desired accuracy, hence conserving resources like bandwidth and energy.

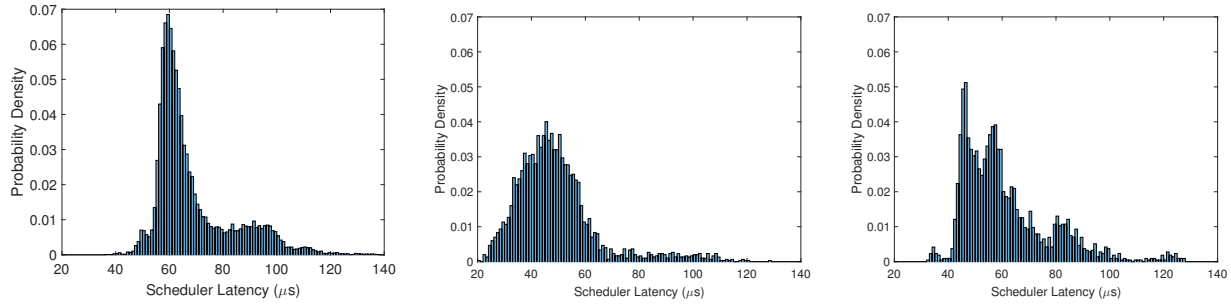


(a) *Timeline* 1 synchronization accuracy

(b) *Timeline* 2 synchronization accuracy

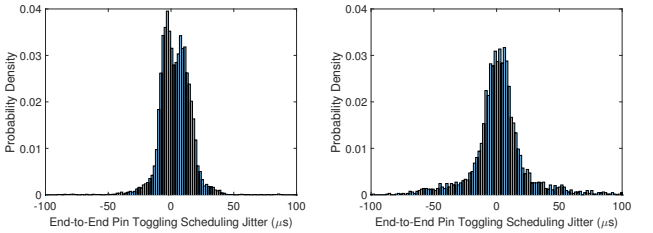
Fig. 10: (a) shows pair-wise error probability density of three nodes *a*, *b*, *c* bound to timeline 1 in Figure 3b with 100 μsec accuracy requirement, (b) shows pair-wise error probability density of three nodes *c*, *d*, *e* bound to timeline 2 with 1 μsec accuracy (Note that x-axis units are in nanoseconds, and x-axis scale changes in (a) and (b)). Note that *c* maintains mappings of both timelines, and the achieved accuracy for all the nodes is almost equal to their desired accuracy

In Figure 9, we show the QoT stack's ability to estimate the uncertainty in synchronization and expose it. Uncertainty captures the variance in time introduced by various sources of errors, that cause the time to deviate from its true value. The red plot provides the ground truth i.e, the actual uncertainty between the *local* timeline reference and the *global* timeline reference, $e = t_{global} - t_{local}$, whereas the green plot is an upper bound on uncertainty estimated by the stack, $e_u = t_{upper} - t_{local}$ and the blue plot is the lower bound on uncertainty estimated by the stack, $e_l = t_{lower} - t_{local}$. Note that the bounds are valid that is, $t_{upper} > t_{global}, t_{local} > t_{lower}$, only when, $e_u > e > e_l$, which is what is achieved in Figure 9. The uncertainty bounds estimated by the stack are applicable, both when synchronization is running or not. Figure 9a shows the bounds when the synchronization is running. Note that these bounds tend to increase when we turn the synchronization off (Figures 9b and 9c). The bounds extend in both directions as



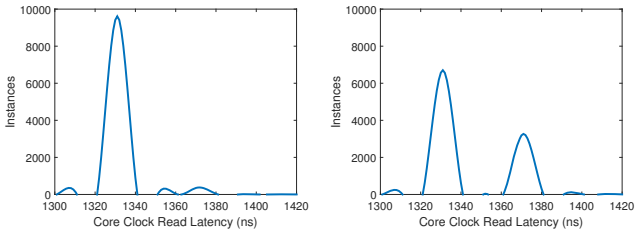
(a) Estimated QoT Scheduler Latency (b) Measured QoT Scheduler Latency (c) Measured Linux RT Scheduler Latency

Fig. 11: Scheduler Latency Distributions, for a periodic pin toggling application on a single node



(a) QoT Stack for Linux (b) Linux RT Scheduler with PTP

Fig. 12: End-to-end scheduling jitter distributions for synchronous pin toggling on two nodes



(a) (b)

Fig. 13: Clock read latency histograms in different time intervals, estimated by the system uncertainty estimation service

a function of variance in frequency bias, and they will always bound the actual uncertainty. The longer the period for which the synchronization is off, the higher will be the uncertainty bounds. Thus, the QoT stack not only reports precise time to the applications but also the uncertainty in time with high confidence bounds.

B. Scheduler Uncertainty

We benchmark the QoT Core's scheduling interface against the Linux *Real-Time* (RT) scheduler by using a periodic pin-toggling application which toggles a memory-mapped GPIO pin, at every second boundary. All the following experiments were conducted under identical load conditions for a duration of 3000 seconds, with the pin-toggling application being the highest real-time priority user application in the system. Multiple sporadic tasks with lower real-time priorities, which used the QoT stack functionality, were also running on the same system.

To measure scheduler uncertainty, we devise the following experiment. On a single node, an application periodically calls the **`timeline_waituntil_nextperiod`** API call, such that the pin toggle is scheduled at every second boundary on a timeline. When the task wakes up, the QoT stack provides a timestamp (with uncertainity) for when the event was actually scheduled. The scheduler latency can be estimated by taking the difference of the timestamps: when the task was supposed to wake up, and when it was actually scheduled. We also empirically measure the scheduler latency by using a Salae Logic Pro 16 logic analyzer [23]. The logic analyzer measures the latency for each pin toggle event by comparing against a deterministic PWM with edges at every second boundary on a timeline reference.

Figure 11a plots the distribution of the scheduler latency as estimated by the QoT stack, while Figure 11b shows the empirically-measured distribution. Observe that the empirical distribution and the distribution provided by our stack share similar characteristics. Thus, the uncertainty estimate provided by the QoT stack holds up to empirical measurement.

For the Linux RT scheduler, using real-time priority scheduling (**`SCHED_FIFO`**), Figure 11c shows the empirically-measured latency distribution, where the **`clock_nanosleep`** system call was used to schedule a periodic pin toggle. Note that the QoT-aware Linux scheduler and the Linux RT scheduler share similar statistical properties. The QoT-aware scheduler provides adherence to our *timeline*-driven architecture with no significant overhead.

The ability to perform choreographed scheduling is key to our stack, and hence we next characterize the end-to-end synchronous scheduling jitter. In our setup, we have two identical applications running on separate nodes. Both applications bind to the same timeline and synchronize with each other. Using the **`timeline_waituntil_nextperiod`** API call, the applications synchronously toggle a memory-mapped GPIO pin at every second boundary on the timeline reference. The synchronization service is also running on both nodes. In Figure 12a, we plot the distribution of the end-to-end jitter between the pin toggles of the distributed application. The instants at which the pins toggled were captured by a logic analyzer, and the difference in timestamps was used to compute the obtained distribution.

We conduct a similar experiment using the Linux **clock_nanosleep** system call on two distributed nodes synchronized by PTP. Figure 12b plots the distribution of the end-to-end scheduling jitter for Linux and PTP. Our stack runs a patched PTP synchronization service, and hence the distribution obtained has a similar jitter profile to that obtained using PTP. Note that our interface is policy-agnostic and does not incur additional overhead, while at the same time providing a range of QoT-based functionality. However, the scheduling jitter can be reduced using more suitable policies in the kernel.

Figure 13 shows two histograms for the estimated latency in reading the core clock from userspace, over different one-second durations, as estimated by the system uncertainty estimation service. Observe that the distributions change over time and are a function of system load. Each peak in the distribution corresponds to different *locks* which cause contention in reading the core clock. This measured distribution plays a key role in continuously keeping track of the uncertainty introduced by the OS in reading the clock.

VIII. Conclusion & Future Work

The timeline abstraction with its associated notion of Quality of Time (QoT) helps virtualize time-related resources in a system, and plays a role analogous to that of sockets with associated Quality of Service (QoS) bindings in network stacks. QoS-aware networking applications can read, write, open and close sockets, and specify QoS parameters. Similarly, QoT-aware time-sensitive applications can bind and unbind from timelines, read and schedule events on the timeline reference, and specify QoT requirements. We make QoT visible and controllable in our timeline-driven architecture. This enables QoT-aware applications to specify their timing requirements, while the system manages clocks and synchronization protocols to provide the appropriate levels of QoT. In the future, this architecture would be extended to address challenges introduced by multiple processing cores, hardware accelerators and peripherals.

Our initial implementation of the QoT Stack for Linux delivers most of our early goals. However, it presently takes advantage of only the *accuracy* attribute of timelines. Future implementations of our stack will also make use of the *resolution* attribute and provide the ability to dynamically switch between hardware clocks based on application requirements. We also plan to support multiple network interfaces, and different oscillators which could be adjusted in hardware. The stack could then switch between different core clocks, use different NICs across heterogeneous networks, and use different synchronization protocols, to best strike a balance between desired performance and resource consumption. Finally, a co-optimization of timelines and synchronization sessions would help conserve network and system resources.

The QoT Stack for Linux is open-source and under development. In the future we plan to support multiple hardware platforms. The code repository can be found at, *https://bitbucket.org/rose-line/qot-stack/src*

Acknowledgment

This research is funded in part by the National Science Foundation under awards CNS-1329755 and CNS-1329644. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, or the U.S. Government

References

[1] B. L. G. Regula, "Formation control of a large group of UAVs with safe path planning," in *IEEE 21st Mediterranean Conference on Control & Automation (MED)*, 2013.

[2] P. V. Estrela and L. Bonebakker, "Challenges Deploying PTPv2 in a Global Financial Company," in *Intl. IEEE Symposium on Precision Clock Synchronization for Measurement & Communication (ISPCS)*, 2012.

[3] S. Natarajan and A. Ganz., "SURGNET: An Integrated Surgical Data Transmission System for Telesurgery," in *International Journal of Telemedicine and Applications Volume, Article ID 435849.*, 2009.

[4] M. Lipiński, T. Włostowski, J. Serrano, and P. Alvarez, "White rabbit: a ptp application for robust sub-nanosecond synchronization," in *Intl. IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2011*. IEEE, 2011.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Googles globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[6] K. Lee, J. C. Eidson, H. Weibel, and D. Mohl, "Ieee 1588-standard for a precision clock synchronization protocol for networked measurement and control systems," in *Conference on IEEE*, vol. 1588, 2005, p. 2.

[7] J. Lundelius and N. Lynch, "An upper and lower bound for clock synchronization," *Information and control*, vol. 62, no. 2, 1984.

[8] H. Zhou, C. Nicholls, T. Kunz, and H. Schwartz, "Frequency accuracy & stability dependencies of crystal oscillators," *Carleton University, Systems and Computer Engineering, Technical Report SCE-08-12*, 2008.

[9] M. Kuperberg and R. Reussner, "Analysing the fidelity of measurements performed with hardware performance counters," in *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*.

[10] T. Broomhead, L. Cremean, J. Ridoux, and D. Veitch, "Virtualize everything but time." in *OSDI*, vol. 10, 2010, pp. 1–6.

[11] K. Lee, D. Chu, E. Cuervo, J. Kopf, A. Wolman, Y. Degtyarev, S. Grizan, and J. Flinn, "Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming," *GetMobile: Mobile Computing and Communications*, vol. 19, no. 3, pp. 14–17, 2015.

[12] "https://beagleboard.org/black," Beaglebone Black.

[13] D. L. Mills, "Internet time synchronization: the network time protocol," *Communications, IEEE Transactions on*, vol. 39, no. 10, 1991.

[14] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[15] P. Derler, T. H. Feng, E. A. Lee, S. Matic, H. D. Patel, Y. Zheo, and J. Zou, "Ptides: A programming model for distributed real-time embedded systems," DTIC Document, Tech. Rep., 2008.

[16] J. Zou, S. Matic, and E. A. Lee, "Ptidyos: A lightweight microkernel for ptides real-time systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*. IEEE, 2012.

[17] B. R. Hamilton, X. Ma, Q. Zhao, and J. Xu, "ACES: Adaptive Clock Estimation and Synchronization Using Kalman Filtering," in *MobiCom*.

[18] X. Xu, "A New Time Synchronization Method for Reducing Quantization Error Accumulation Over Real-Time Networks: Theory and Experiments Evaluation of kalman filtering for network time keeping," in *IEEE Trans. on Industrial Informatics*, 2013.

[19] "http://pubs.opengroup.org/onlinepubs/009695399/functions/clock.html," IEEE Standard 1003.1, 2004.

[20] "dds for real-time system, prismtech," OpenSplice.

[21] "http://linuxptp.sourceforge.net," The Linux PTP Project.

[22] "http://w3.siemens.com," Siemens RUGGEDCOM RSG2488.

[23] "http://downloads.saleae.com," Salae Logic Pro 16, 2015.

APPENDIX

A. Timelines and Bindings

Timelines are ordered on a red-black tree (left in figure below), and bindings as two parallel sorted linked lists (right in figure below) – one representing accuracy and the other representing resolution. Of greatest importance to time management and synchronization is the ability to calculate the dominating accuracy and resolution for any given timeline, which is the highest accuracy and resolution across all associated bindings. We maintain two parallel linked lists of bindings - ordered by increasing accuracy and resolution respectively, against a single timeline node in the red-black tree. This way, we can efficiently track the dominating accuracy and resolution in a manner that is robust to the addition of a new binding, removal of an existing binding, and update of accuracy and resolution of an existing binding. In the figure below, Timeline "T2" has two bindings: Binding 1 has the dominant accuracy requirement, while Binding 2 has the dominant resolution requirement.

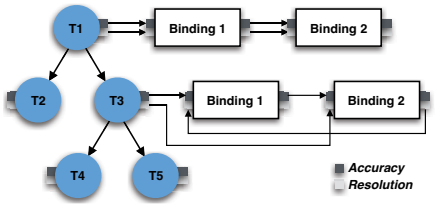


Fig. 14: Sorting Timelines and Bindings

B. QoT Stack Data Types

Listing 2: Time Data Types

```
/* A duration of time */
typedef struct timelength {
  u64 sec;    /* Seconds */
  u64 asec;   /* Fractional seconds in attoseconds */
} timelength_t;

/* A single point in time */
typedef struct timepoint {
  s64 sec;    /* Seconds since reference */
  u64 asec;   /* Fractional seconds in attoseconds */
} timepoint_t;

/* An interval of time */
typedef struct timeinterval {
  timelength_t below;     /* Seconds below (-ve) */
  timelength_t above;     /* Seconds above (+ve) */
} timeinterval_t;

/* A point in time with an interval of uncertainty */
typedef struct utimepoint {
  timepoint_t estimate;     /* Estimate of time */
  timeinterval_t interval;  /* Uncertainty interval */
} utimepoint_t;

/* A duration of time with an uncertain end point */
typedef struct utimelength {
  timelength_t estimate;    /* Estimate of time */
  timeinterval_t interval;  /* Uncertainty interval */
} utimelength_t;
```

Listing 3: Timeline Data Types

```
/* Timeline Information */
typedef struct qot_timeline {
    char name[QOT_MAX_NAMELEN]; /* Timeline name       */
    int index;                  /* Timeline index      */
    struct rb_root event_head;  /* Events RB tree head */
    raw_spinlock_t rb_lock;     /* RB tree spinlock    */
} qot_timeline_t;

/* QoT comprised of (min, max) accuracy and resolution */
typedef struct timequality {
  timelength_t resolution;  /* Time resolution */
  timeinterval_t accuracy;  /* Time accuracy   */
} timequality_t;

/* Binding Information */
typedef struct qot_binding {
    char name[QOT_MAX_NAMELEN];  /* Application name */
    timequality_t demand;        /* Requested QoT    */
    int id;                      /* Binding ID       */
} qot_binding_t;

/* Timeline Type */
typedef struct timeline {
    qot_timeline_t info;  /* Timeline information    */
    qot_binding_t binding;/* Binding information     */
    int fd;    /* File descriptor to /dev/timelineX */
    int qotusr_fd;/* File descriptor to /dev/qotusr */
} timeline_t;
```

Listing 4: Clock Data Type

```
/* QoT Clock (admin only) */
typedef struct qot_clock {
    char name[QOT_MAX_NAMELEN];  /* Clock name         */
    qot_clk_state_t state;       /* Clock state        */
    u64 nom_freq_nhz;            /* Frequency in nHz   */
    u64 nom_freq_nwatt;          /* Power in nWatt     */
    utimelength_t read_lat;      /* Read latency       */
    utimelength_t interrupt_lat; /* Interrupt latency  */
    u64 errors[QOT_CLK_ERR_NUM]; /* Error properties   */
    int phc_id;                  /* PTP device ID      */
} qot_clock_t;
```

C. Precision Time Protocol

Precision Time Protocol (PTP) [6] is an IEEE 1588 compliant time synchronization protocol and it synchronizes clocks over a multicast capable network. It provides a best master clock algorithm that identifies the best clock in the network and choose it as a master. A slave clock synchronizes to the master clock using bidirectional communication. The master sends a 'Sync' packet along with the timestamp when the packet left the master node. The slave receives the 'Sync' packet and timestamps its arrival time. The slave determines its clock offset from the master by calculating the difference in 'Sync' packet's departure and arrival times, and adjusts its Network Interface Clock (*/dev/ptpX*) accordingly. The slave also compensates for the network propagation delay by exchanging 'delay request' and 'delay response' packets with the master.

D. QoT Application Programming Interface

In this section, we contrast the QoT-aware TDMA application (Listing 1) with the TDMA application using existing Linux POSIX APIs (Listing 5). The Linux API based TDMA application does not have a notion of QoT and cannot provide end-to-end estimates on timing uncertainty. The application computes its wake up time every period, and uses the **`clock_nanosleep`** system call to wake up and transmit a

TABLE II: Quality of Time APIs

Category	API	Return Type	Functionality
Timeline Association	**timeline_bind** (name, accuracy, resolution)	timeline	Bind to a timeline
	timeline_unbind (timeline)	status	Unbind from a timeline
	timeline_getaccuracy (timeline)	accuracy	Get binding accuracy
	timeline_getresolution (timeline)	resolution	Get Binding resolution
	timeline_setaccuracy (timeline, accuracy)	status	Set Binding accuracy
	timeline_setresolution (timeline, resolution)	status	Set Binding resolution
Time Management	**timeline_gettime** (timeline)	uncertain_timestamp	Get timeline reference time with uncertainty
	timeline_getcoretime ()	uncertain_timestamp	Get core time with uncertainty
	timeline_core2rem (timeline, core_time)	uncertain_timestamp	Convert a core timestamp to a timeline reference
	timeline_rem2core (timeline, time)	uncertain_timestamp	Convert a timeline reference timestamp to core time
Event Scheduling	**timeline_waituntil** (timeline, absolute_time)	uncertain_timestamp	Absolute timed wait
	timeline_sleep (timeline, interval)	uncertain_timestamp	Relative timed wait
	timeline_setschedparams (timeline, period, start_offset)	status	Set period and start offset
	timeline_waituntil_nextperiod (timeline)	uncertain_timestamp	Absolute timed wait until next period
	timeline_timer_create (timeline, period, start_offset, count, callback)	timer	Register a periodic callback
	timeline_timer_cancel (timer)	status	Cancel a periodic callback
	timeline_config_events (timeline, event_type, event_config, enable, callback)	status	Configure events/external timestamping on a pin

packet. If the timing uncertainty exceeds the guard bands, then packets will collide. On the other hand, a QoT-aware TDMA application can take a decision on packet transmission based on the returned QoT.

Alternatively, a developer may create a daemon to compute the end-to-end timing uncertainty. However, this involves significant effort, complex interactions with existing timing systems, and privileged system access. Given that this functionality is commonly required across a range of applications, our stack provides it as a system service.

Listing 5: TDMA Application using Linux API

```
clock_gettime(CLOCK_REALTIME, now); /* Get time */
/* Current TDMA cycle number and start time */
int tdma_cycle_no = timespec2ns(now)/TDMA_PERIOD;
uint64_t cycle_start_ns = tdma_cycle_no*TDMA_PERIOD;
/* Slot time offset from start of cycle */
uint64_t slot_offset = get_my_slot()*TDMA_SLOT_LENGTH;
uint64_t transmit_time_ns = cycle_start_ns + slot_offset;
/* Periodic TDMA Transmission */
while (tdma_running) {
  /* Time of next transmission */
  transmit_time_ns = transmit_time_ns + TDMA_PERIOD;
  /* Sleep till transmit slot. Also handle signals */
  while(clock_nanosleep(CLOCK_REALTIME, ns2timespec(↩
     transmit_time_ns) == EINTR);
  clock_gettime(CLOCK_REALTIME, now); /* Get time */
  /* Transmit if wakeup time within bound */
  /* Packets may collide due to bad sync or clock */
  if(timespec_compare(now, transmit_time_ns + ↩
     TDMA_GUARD_BAND) <= 0 ) {
    transmit_packet(message);
  } else {
    hold_off();
  }
}
```

E. Beaglebone Black QoT Clock Drivers

To support the QoT stack functionality, the following clock drivers were implemented for the Beaglebone Black platform:

TI CPSW Network Interface Clock: The Linux kernel already ships with TI's Common Platform Ethernet Switch (CPSW) Drivers (which also supports the AM335x SoC), which can be found in the kernel at **Linux/drivers/net/ethernet/ti**. The Common Platform Time Stamping (CPTS) module inside the CPSW ethernet subsystem is used to facilitate host control of time synchronization related operations. CPTS supports ethernet receive events, ethernet transmit events, and hardware and software timestamp push events. By default, hardware timestamp push events are disabled in the CPTS module, so we patched it and enabled time stamp inputs (HW1/4_TS_PUSH) to load the timestamp push events into the FIFO. These time stamp inputs can be triggered from Timers 4-7 of the AM335X. This setup enables NIC to Core synchronization (described in Section IV-B) with high accuracy.

BBB-AM335x Core Clock: This driver makes use of the on board dual mode timers (**dmtimers**) to provide various core clock related functionality. The AM335x contains 7 timers. All of them can be driven by on board oscillators, and some provide external clocking ability. Timers 1 and 2 are already used by the Linux kernel, hence, we use timers 3-7 to demonstrate the range of functionalities of a core clock. All the timers are clocked by an onboard 24 MHz crystal. The timers and their corresponding function are as follows:

- **Timer 3:** Drives the monotonic core clock.
- **Timer 4:** Enables scheduling, by providing the ability to trigger interrupts in the future.
- **Timer 5:** Generates a precise Pulse-per-second (PPS) which is used to discipline the Network Interface Clock.
- **Timer 6:** Provides the ability to timestamp external events on a pin.
- **Timer 7:** Provides the ability to generate a periodic output on a pin.

In order to deliver the functionality of timers 5-7 the system needs to configure the GPIO pins. For ARM based platforms, this is done by using a device tree. A device tree enables a user to configure the peripherals of an embedded ARM platform at run-time. Our prototype stack also provides a device tree for the Beaglebone Black platform.